Parson's Problems Appliance for Learning Management Systems (PPALMS)

Design Document

Authors

Anthony Narlock
Stephanie Ye
Shen Lua
Jaden Rodriguez

Group 2

This page intentionally left blank.

Document Revision History

Date	Version	Description	Author
11/06/2022	v0.0.1	Initial draft	Anthony Narlock Shen Lua Stephanie Ye Jaden Rodriguez

This page intentionally left blank.

Contents

1 Introduction

1.1 Definitions, Acronyms, and Abbreviations

The following list contains the necessary definitions, acronyms, and abbreviations that are used throughout this document. The software engineer must understand these definitions.

Acronyms:

GUI : Graphical User Interface

JAR : Java ARchive

LMS : Learning Management System

MVC : Model-View-Controller

PPALMS : Parson's Problems Appliance for Learning Management Systems

SRS : Software Requirements Specification

JSON : JavaScript Object Notation

Definitions:

User : Term for person using the PPALMS application (e.g. instructors, researchers, teaching assistants)

1.2 Purpose

The purpose of this design document is to outline design details for software engineers to implement the PPALMS system. This document details an object-oriented implementation satisfying the PPALMS requirements

1.3 System Overview

The Parson's Problems Appliance for Learning Management Systems (PPALMS) is intended for users who want to create Parson's Type problems. A goal of PPALMS is to make Parson's Problems more accessible and frequent in academic testing. This system would appeal to environments like Universities, Learning Management System businesses, and testing/merit corporations like College Board who specialize in academic testing services. It would be expected to be used by test-makers like professors, instructors, and teaching assistants.

1.4 Design Objectives

A primary design goal is ensuring that **performance** is adequate so as to not hinder the User's experience. The **performance** of the PPALMS system can be broken down into three main sections: *response time*, *throughput*, and *memory requirements*.

With regards to *response time* of the PPALMS, general interaction with the GUI should be short in duration, so the user has positive feedback. User actions such as data entry during the problem creation process need to be less than a couple hundred milliseconds. Trivial tasks such as uploading the source code file or submitting the problem creation form should all complete within one second. For more intensive tasks such as the final export process, the time required for completion would vary and is more dependent on factors such as a stable network connection, external server speeds, and the amount of processing power the user's machine has.

With regards to *throughput*, it is not a significant limitation for PPALMS because the problem size has been constrained, and the speed at which it is being constructed by the User is slow compared to any computer which would run this application.

With regards to *Memory requirements* for the PPALMS, since the User is expected to be a general instructor, the apparatus required to use the application should be minimal. The PPALMS can occupy no more than a gigabyte of disk space, and require no more than 500 megabytes of RAM at any time.

Moreover, the PPALMS application must be **dependable** at all times. It must be *robust* in the sense that it may accept various readable text format source code files, and generates an appropriate error message if the file is not text-based or exceeds the size constraint. It must be *reliable* whereby each user input is validated by the controller and confirmed in the view. *Availability* via system requirement checks will be made to ensure that the application will run smoothly given normal circumstances. *Security* of the PPALMS is integral so as to not allow for malicious intent. This is circumvented by ensuring the input fields such as the file import function, and form would have filters and checks in place and only allow for valid inputs to be accepted. And finally, *safety*. Based on the PPALMS requirement and specification documents, the PPALMS application should operate and function as stated and not beyond its required capabilities. This is for safety.

The **cost** of designing, developing, and deploying the PPALMS is a long and extensive process. It's important that we take into account costs in defining whether certain features or functions are achievable. The Proper **maintenance** of the PPALMS application is important to the longevity of the system. By using the MVC architecture and OO, this implementation has been separated into modules, thus giving it low coupling and high cohesion. Hence it will be *extensible* and *modifiable* because it will be easy to add and change functionality after initial deployment of the system. PPALMS will be *adaptable* and *portable* because of its dependency on the Java runtime environment and acceptance of various file and LMS services. This will allow it to run on different architectures, edit a variety of content, and deliver to different services. This design is *traceable* because the design was constructed to satisfy the requirements specification.

Finally, the **End-User** goal focuses on the *utility* of the PPALMS application, given that if the user uploads a valid text file of source code with no more than 50 lines of code, the PPALMS should function as intended. And from a *usability* perspective, the PPALMS application with its minimal and structured GUI written in Java Swing, should have an easily navigable user experience.

1.5 *References*

The following list contains all of the referenced material that has been used to create this document.

- IEEE Software Design Document Template,
- PPALMS Software Requirement Specification Document v0.0.2
- Java Swing - for the creation of the Sample GUI provided in Section 3.1
- Code Conventions for the Java Programming Language.

2 Design Overview

2.1 *Introduction*

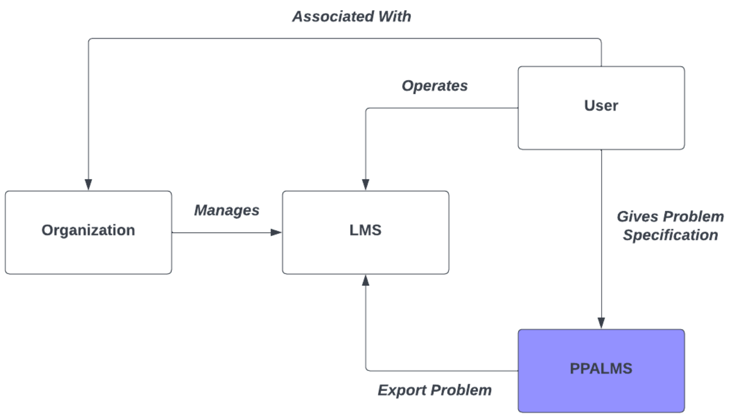
The PPALMS system will have an object-oriented design with a MVC architecture. There are three components that make up the MVC architecture: The Model, which is the central component of the system that provides the application's primary data structure and allows for operations like retrieving and manipulating; The View, which serves as the physical representation that the user can see; and The Controller, which takes input commands and updates the model or view. In this implementation of the PPALMS application, the Model is the Parson's problem being designed by the User, The View is the graphical user interface that the User will be able to interact with, and The Controller will dynamically update the problem as the User edits the provided input fields.

The tools *LucidChart* and *Draw.io* were used to create block diagrams.

The following subsections provide more details about the design overview and give the software engineer more insight into the design of the application.

2.2 *Environment Overview*

The PPALMS software application will be independent of the other systems that will use it. In the SRS document, we provided a high-level context diagram to help give a better understanding of how the PPALMS application interacts with the world.



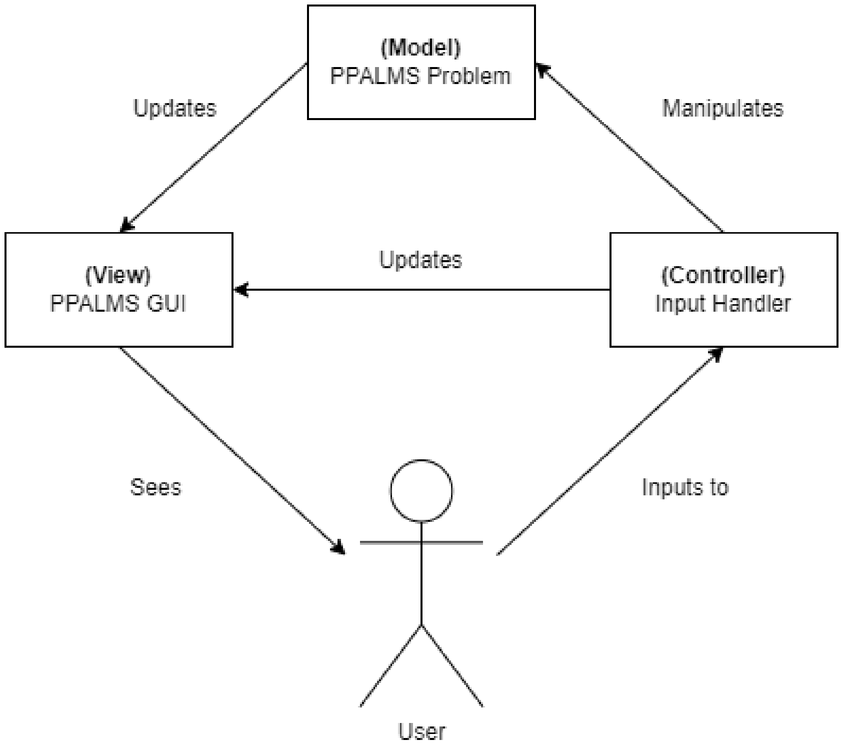
Context Diagram from SRS v0.0.2. *PPALMS* and *LMS* are systems.

The PPALMS application will be run by a User on a computing device that has a storage apparatus, like a hard drive, since the application will need to take in a user's source code. Because the PPALMS application will have a GUI, the computing device will have a display output like a monitor so the user will be able to see the actions that they are performing. User's will perform their actions (e.g. upload source code, title and annotate a problem) using peripheral devices that are connected to the computing device like a keyboard and mouse. The PPALMS application will be implemented using the Java programming language and will utilize the Java Swing library for creating the GUI. The application will reside entirely on the User's machine as an executable JAR file.

The PPALMS system will interact with external machines only when it is exporting problems. A management organization will be able to interact with User-made problems once PPALMS successfully exports it to a target LMS.

2.3 System Architecture

The following section describes an overview of the system architecture as described in Section 2.1.



The figure above describes the architectural view of the PPALMS system. The PPALMS system consists of three major components, all of which highlight the use of the MVC design pattern. Firstly, we have the View, which serves as the GUI that the user will be able to see physically on their computer screen. On this GUI, the user will be able to directly interact with the Controller by providing inputs based on what they are able to see from the View. Examples of these can be found in Section 5.

Since the PPALMS system application is a stand-alone application, it only contains this one system design. The application will be controlled by a single system and does not contain any other subsystems.

2.4 Constraints and Assumptions

One major limiting constraint will be the problem types supported by each LMS system. Each LMS system has its own supported problem types, and a user must be limited to only be able to annotate problems supported by their target LMS. We chose to limit problem types given the LMS system they choose so that the user does not have to go through the entire process of annotating before realizing that their desired problem type is not supported.

One design choice we made was to export the Parson's Problems as a .json file in order to store the optional title, description, and permutations of the problem as arrays. Here, we are making an assumption that the target LMS supports .json files but what the user wants to do with the resulting output file is up to them and is therefore outside the scope of our system.

The PPALMS system's interface language is in English, therefore, the users and developers must be able to speak English. The PPALMS application will be implemented using the Java programming language (Version 8). This means that the developer must have a Java environment ready for development. This includes installing a Java Development Kit that supports Java Version 8. The developer must have knowledge of the Java Swing library for implementing the GUI. For developer implementation conventions, the PPALMS application uses standard Code Conventions for the Java Programming Language.

The PPALMS system would be expected to have new versions put out regularly as new supported LMS systems are added. Hence, a user may need to update frequently.

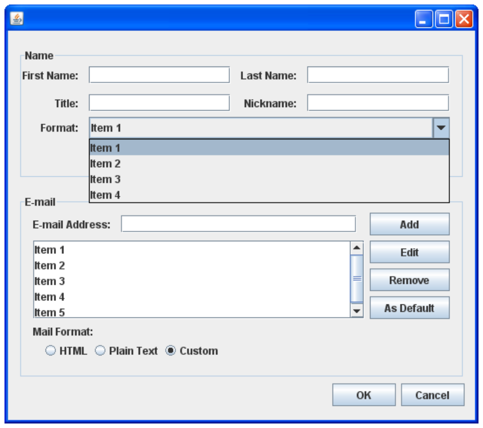
3 Interfaces and Data Stores

3.1 System Interfaces

This section documents the various interfaces provided to users and/or other external systems. Previously we had included user interface description in the SRS document so here we are recapturing them again. No interfaces to other systems are provided as PPALMS is independent of the other systems that will use it.

3.1.1 Application Interface

This interface is used to display and read in input options. The GUI shall have various buttons for different steps in inputting to the PPALMS, one of which is the Upload button for uploading source code. Another button is the Export button for when all required input fields have been filled out and the user is ready to generate the Parson's Problems and export them to the target LMS. The problems that are generated will be stored in files, which is a type of data store. In addition to buttons, the GUI shall have drop down menus for selection options at other steps in inputting to the PPALMS. A drop down menu shall be used for selecting the target LMS and also for selecting the Parsons Problem type. Finally, there shall also be text-entry boxes for if the user wants to input an optional title and description for their Parsons Problems and also for where the source code file shall be displayed for annotations to be made. Below is an example image of a very bare bones Java GUI which includes examples for buttons, drop down menu, and text-entry boxes.



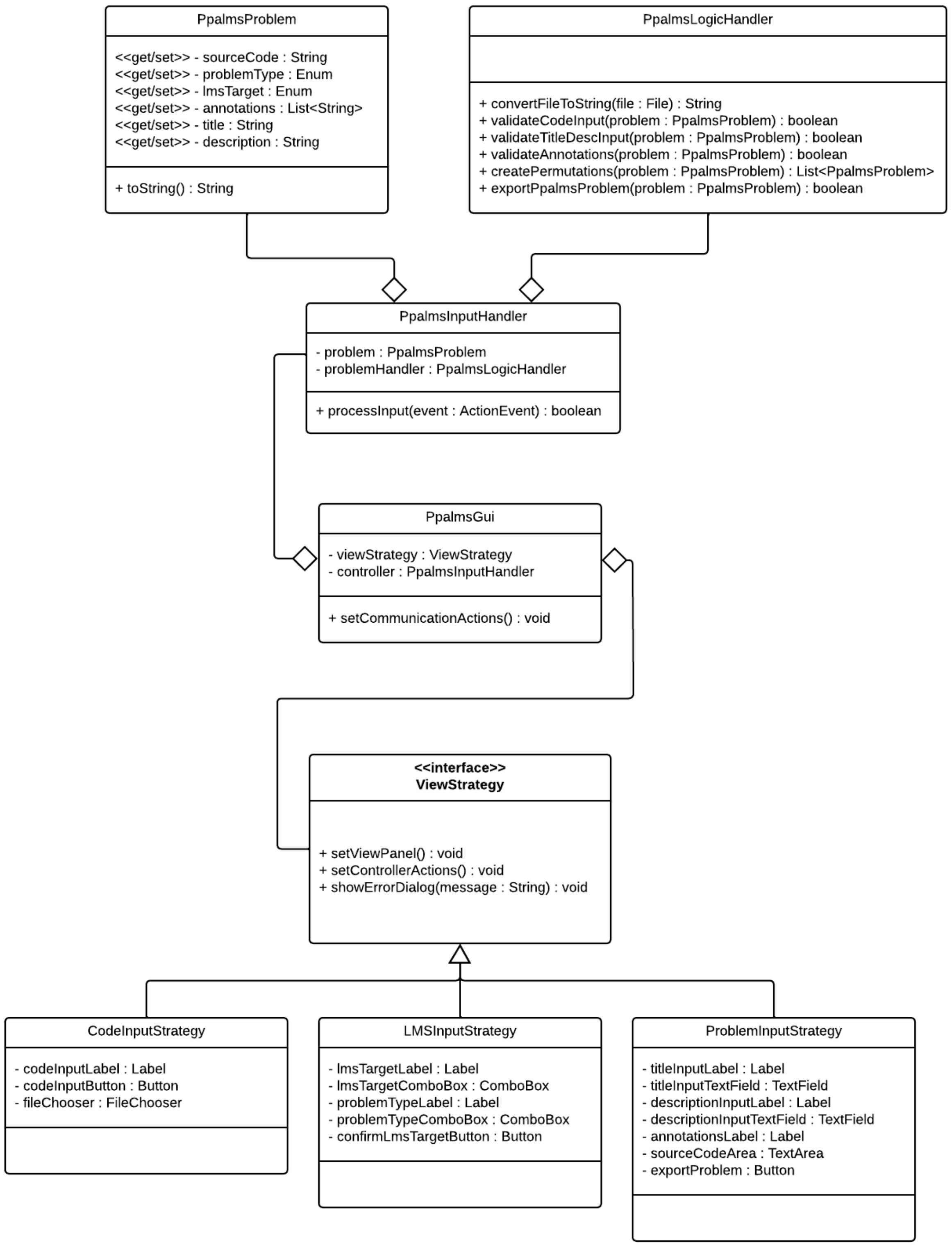
A sample user interface utilizing Java Swing

3.2 Data Stores

PPALMS will need a form of a data store, like a file, for keeping track of the generated problems. We were thinking of using .json file type in order to store the optional title, description, and permutations of the problem as arrays. What the user wants to do with the resulting output file is up to them and is therefore outside the scope of our system. We assume that the target LMS supports the data store we export, which in this case will be a JSON file.

4 Structural Design

4.1 Class Diagram


PpalmsProblem
<<get/set>> - sourceCode : String
<<get/set>> - problemType : Enum
<<get/set>> - lmsTarget : Enum
<<get/set>> - annotations : List<String>
<<get/set>> - title : String
<<get/set>> - description : String
+ toString() : String
PpalmsLogicHandler
+ convertFileToString(file : File) : String
+ validateCodeInput(problem : PpalmsProblem) : boolean
+ validateTitleDescInput(problem : PpalmsProblem) : boolean
+ validateAnnotations(problem : PpalmsProblem) : boolean
+ createPermutations(problem : PpalmsProblem) : List<PpalmsProblem>
+ exportPpalmsProblem(problem : PpalmsProblem) : boolean
PpalmsInputHandler
- problem : PpalmsProblem
- problemHandler : PpalmsLogicHandler
+ processInput(event : ActionEvent) : boolean
PpalmsGui
- viewStrategy : ViewStrategy
- controller : PpalmsInputHandler
+ setCommunicationActions() : void
<<interface>>
ViewStrategy
+ setViewPanel() : void
+ setControllerActions() : void
+ showErrorDialog(message : String) : void
CodeInputStrategy
- codeInputLabel : Label
- codeInputButton : Button
- fileChooser : FileChooser
LMSInputStrategy
- lmsTargetLabel : Label
- lmsTargetComboBox : ComboBox
- problemTypeLabel : Label
- problemTypeComboBox : ComboBox
- confirmLmsTargetButton : Button
ProblemInputStrategy
- titleInputLabel : Label
- titleInputTextField : TextField
- descriptionInputLabel : Label
- descriptionInputTextField : TextField
- annotationsLabel : Label
- sourceCodeArea : TextArea
- exportProblem : Button

The figure above indicates the structural class diagram of the PPALMS application. This diagram contains all of the classes that the PPALMS system is intended to contain. It is organized in such a way where we can clearly see how we make use of the MVC architecture. To continue this narrative, the top two classes, PpalmsProblem and PpalmsLogicHandler are the classes that make up the Model of the system. The PpalmsInputHandler serves as the Controller for the PPALMS application. This class will be handling input from the user and sending it to the model to be manipulated. Next, we have the View which is represented by the PpalmsGui class (the GUI). This is the class that will implement the visual interface that the user will interact with. Specifically, this class contains a ViewStrategy which will determine the behavior of what the GUI will be displaying based on the sequence of events the user will undergo while using the PPALMS application. CodeInputStrategy, LMSInputStrategy, and ProblemInputStrategy are all concrete implementations of the ViewStrategy, which all share their own display functions the user will interact with. The following sections provide more detail of these classes.

4.2 Class Descriptions

4.2.1 Class: PpalmsProblem

- Purpose: *Represents the Parson's problem being designed*
- Constraints: *None*
- Persistent: *Yes, is used throughout problem creation process*

4.2.1.1 Attribute Descriptions

- Attribute: *problemType*
  - Type: Enum
  - Description: *The problem's type chosen by User*
  - Constraints: *Must be member of type Enum*
- Attribute: *sourceCode*
  - Type: String
  - Description: *The problem's source code uploaded by User*
  - Constraints: *Must not exceed 50 lines*
- Attribute: *lmsTarget*
  - Type: Enum
  - Description: *The name of the chosen target LMS*
  - Constraints: *Must be a member of LMS Enum .*
- Attribute: *annotations*
  - Type: List<String>
  - Description: *List of annotations made by the user*
  - Constraints: *None*
- Attribute: *title*
  - Type: String
  - Description: *Optional string title of problem being designed*
  - Constraints: *None*

- Attribute: *description*
  - Type: String
  - Description: *Optional string describing the problem being designed*
  - Constraints: *None*

4.2.1.2 Method Descriptions

- Method: *toString*
  - Return Type: *String*
  - Parameters: *None*
  - Return value: *String representation of Parson's problem being designed*
  - Pre-condition: *The* PpalmsProblem *has a non-empty annotations list*
  - Post-condition: *A string representation which can be used in the problem creation process is created*
  - Attributes read/used: *sourceCode, annotations*
  - Methods called: *Trivial Getters/Setters of attributes.*
  - Test Cases:
    - *Ensure successful representation is made upon call. Expected output is dependent on the concrete implementation of the* PpalmsProblem*, but should correspond to its type. This method should be pure and not leave artifacts.*
    - *Ensure failure if annotations have not been made.*

4.2.2 Class: PpalmsLogicHandler

- Purpose: *To validate and manage data from upload, throughout the design process, and to completion of exportation*
- Constraints: *None*
- Persistent: *Yes, is used throughout problem creation process*

4.2.2.1 Attribute Descriptions

There are no attributes for this class.

4.2.2.2 Method Descriptions

- Method: *convertFileToString*
  - Return Type: *String*
  - Parameters: *File*
  - Return value: *String with contents of source code File*
  - Pre-condition: *The User has chosen a file via the CodeInputStrategy*
  - Post-condition: *A string representation which can be used in problem design is created*
  - Attributes read/used: *None*
  - Methods called: *None*
  - Test Cases:
    - *Ensure string storing file contents is successfully made upon call. This method should be pure and not leave artifacts.*

        - *Ensure failure if the file contains non-human-readable bytes. Problem creation should be interrupted and the User should be signaled that their chosen file contains unreadable characters.*
- Method: *validateCodeInput*
    - Return Type: *boolean*
    - Parameters: *PpalmsProblem*
    - Return value: *Boolean value representing whether problem code satisfies requirements*
    - Pre-condition: *The* PpalmsProblem *sourceCode attribute has been initialized*
    - Post-condition: *Problem creation process may continue uninterrupted*
    - Attributes read/used: *sourceCode*
    - Methods called: *Trivial Getters/Setters of attributes.*
    - Test Cases:
        - *Ensure the sourceCode attribute satisfies the constraints upon a return of True. This method is pure and should not leave artifacts.*
        - *Ensure failure if source code exceeds 50 lines. Problem creation should be interrupted and the User should be signaled that their chosen file exceeds the input limit of 50 lines.*
- Method: *validateTitleDescInput*
    - Return Type: *boolean*
    - Parameters: *PpalmsProblem*
    - Return value: *Boolean value representing whether title and description satisfy requirements*
    - Pre-condition: *None*
    - Post-condition: *The PpalmsProblem title and description have been validated to continue design process*
    - Attributes read/used: *title, description*
    - Methods called: *Trivial Getters/Setters of attributes.*
    - Test Cases:
        - *This method should be successful so long as each attribute has been initialized, even if they are empty. This method should be pure and leave no artifacts.*
        - *Ensure failure if title or description have not been initialized*
- Method: *validateAnnotations*
    - Return Type: *boolean*
    - Parameters: *PpalmsProblem*
    - Return value: *Boolean value representing whether annotations satisfy requirements*
    - Pre-condition: *The* PpalmsProblem *has a non-empty annotations list*
    - Post-condition: *The PpalmsProblem annotations have been validated to continue design process*
    - Attributes read/used: *problemType, annotations,* possibly *sourceCode*
    - Methods called: *Trivial Getters/Setters of attributes.*
    - Test Cases:

        - *Ensure annotations satisfy requirements. Expected output is dependent on the concrete implementation of the* PpalmsProblem*, but should correspond to the annotations satisfying the format of the problem type. This method should be pure and not leave artifacts.*
        - *Ensure failure if annotations have not been made*
- Method: *createPermutations*
    - Return Type: *List<PpalmsProblem>*
    - Parameters: *PpalmsProblem*
    - Return value: *List of different permutations of the input PpalmsProblem*
    - Pre-condition: *The* PpalmsProblem *has been signaled to export*
    - Post-condition: *A list of problem permutations has been created for export*
    - Attributes read/used: *title, description, lmsTarget, sourceCode, annotations (of input)*
    - Methods called: *Trivial Getters/Setters of attributes.*
    - Test Cases:
        - *Ensure permutations list is created successfully. The length should not exceed 30, and each element should have a unique and complete permutation of the annotations.*
        - *Ensure failure if annotations have not been made, if the output list exceeds 30 elements, the elements are not unique, or of the correct length.*
- Method: *exportPpalmsProblem*
    - *Return Type: boolean*
    - Parameters: *PpalmsProblem*
    - Return value: *Boolean representing whether was successfully exported*
    - Pre-condition: *The* PpalmsProblem *has been validated*
    - Post-condition: *The PpalmsProblem has been exported, or failure signaled*
    - Attributes read/used: *sourceCode, annotations*
    - Methods called: *Trivial Getters/Setters of attributes, validation methods, createPermutations*
    - Test Cases:
        - *Ensure output JSON to be given to lmsTarget is constructed properly with title, description, and annotation permutations array, and then sent properly. This should return true.*
        - *Ensure failure if any attribute is invalid or the method of exportation fails. This should return false.*

4.2.3 Class: PpalmsInputHandler

- Purpose: *To process inputs by confirming their validity and making the corresponding records*
- Constraints: *Must confirm each change to the PpalmsProblem with the PpalmsLogicHandler*
- Persistent: *Yes, is used throughout the problem creation process*

4.2.3.1 Attribute Descriptions

- Attribute: *problem*
  - Type: *PpalmsProblem*
  - Description: *The problem subject to design by the User*
  - Constraints: *Must maintain validity according to the handler*
- Attribute: *problemHandler*
  - Type: *PpalmsLogicHandler*
  - Description: *Handler which manages validity throughout input to export*
  - Constraints: *None*

4.2.3.2 Method Descriptions

- Method: *processInput*
  - Return Type: *boolean*
  - Parameters: *ActionEvent*
  - Return value: *Boolean representing input confirmation*
  - Pre-condition: *The User triggers an ActionEvent via the GUI by interacting with a component (Button, TextField, etc.).*
  - Post-condition: *The corresponding change is validated and made to the problem, including exportation.*
  - Attributes read/used: *problem, problemHandler*
  - Methods called: *Trivial getters/setters, LogicHandler validation methods*
  - Test Cases:
    - *Ensure annotation corresponding to the action event is validated and set in problem attribute upon success*
    - *Ensure title and description corresponding to the action event is validated and set in problem attribute upon success*
    - *Ensure target LMS corresponding to the action event is validated and set in problem attribute upon success*
    - *Ensure failure if the ActionEvent corresponds to an invalid annotation or target LMS*

4.2.4 Class: PpalmsGUI

- Purpose: *The graphical user interface that the user will be interacting with.*
- Constraints: *None*
- Persistent: *Yes,* PpalmsGUI *is used throughout all processes in the application.*

4.2.4.1 Attribute Descriptions

- Attribute: *viewStrategy*
  - Type: ViewStrategy *(see Section 4.3.6 for definition)*
  - Description: *The view's state given by a concrete* ViewStrategy
  - Constraints: *Must be a concrete implementation of* ViewStrategy *interface.*
- Attribute: *controller*
  - Type: PpalmsInputHandler *(see Section 4.3.4 for definition)*
  - Description: *The View-to-Controller connection for handling user input.*
  - Constraints: *None*

4.2.4.2 Method Descriptions

- Method: *setCommunicationActions*
  - Return Type: *void*
  - Parameters: *None*
  - Return value: *N/A*
  - Pre-condition: *The* PpalmsGUI *has an initialized* ViewStrategy
  - Post-condition: *Communication actions between View and Controller have been created.*
  - Attributes read/used: *viewStrategy, controller*
  - Methods called: *Trivial Getters/Setters of attributes.*
  - Test Cases:
    - *Ensure successful communications are set between View and Controller. Expected output is dependent on the concrete implementation of the* ViewStrategy. *Upon setting these communications, calling processing inputs via the controller should result in success/failure cases that are described in the* PpalmsLogicHandler *that belongs to the* PpalmsInputHandler.
    - *Ensure failure if View and/or controller are not properly initialized. We expect a failure for non-initialized objects.*

4.2.5 Class: ViewStrategy

- Purpose: *Interface for physical user interface attributes.*
- Constraints: *Belongs to PpalmsGUI.*
- Persistent: *Yes, because PpalmsGUI is persistent.*

4.2.5.1 Attribute Descriptions

There are ***no*** *attributes for ViewStrategy due to the nature of the interface. The concrete implementations provide individual attributes based on what strategy is declared.*

4.2.5.2 Method Descriptions

- Method: *setViewPanel*
  - Return Type: *void*
  - Parameters: *None*
  - Return value: *N/A*
  - Pre-condition: *The* ViewStrategy *has been initialized.*
  - Post-condition: *The* ViewStrategy *has been initialized.*
  - Attributes read/used: *"this"*
  - Methods called: *Trivial Getters/Setters of concrete attributes.*
  - Test Cases:
    - *Ensure successful creation of the view's panel. The visual aspects are initialized and appear on the GUI.*
    - *If some attribute of a concrete implementation is not initialized, then we will expect it to not appear on the GUI. This indicates failure.*
- Method: *setControllerActions*

  - Return Type: *void*
  - Parameters: *None*
  - Return value: *N/A*
  - Pre-condition: *The* ViewStrategy *has been initialized.*
  - Post-condition: *The* ViewStrategy *has been initialized.*
  - Attributes read/used: *“this”*
  - Methods called: *Trivial Getters/Setters of concrete attributes.*
  - Test Cases:
    - *Ensure successful actions through the controller have been defined. This means that the specific attributes of the concrete implementations of the interface are able to be properly tested. For example, if the current viewStrategy is the concrete implementation CodeInputStrategy, then pressing the “codeInputButton” will properly open the fileChooser menu - indicating to the user they can select their source code file.*
    - *If some attribute of a concrete implementation is not initialized, then we will expect it to not have a proper controller action. This indicates failure.*
- Method: *showErrorDialog*
  - Return Type: *void*
  - Parameters: *message (String)*
  - Return value: *N/A*
  - Pre-condition: *The* ViewStrategy *has been initialized.*
  - Post-condition: *The* ViewStrategy *has been initialized.*
  - Attributes read/used: *“this”*
  - Methods called: *Trivial Getters/Setters of concrete attributes.*
  - Test Cases:
    - *The system will need to be able to handle unexpected errors. In the Java programming language, we can define special error/exception handling cases. Under fail cases related to the GUI, we will be creating an error dialog message that will be displayed to the screen.*
    - *If some GUI related activity fails and the PPALMS system does not indicate an error dialog, then this will be a failure for the showErrorDialog method.*

4.2.6 Class: CodeInputStrategy

- Purpose: *View for inputting source code for PPALMS problem.*
- Constraints: *See Section 4.3.6.*
- Persistent: *See Section 4.3.6.*

4.2.6.1 Attribute Descriptions

- Attribute: *codeInputLabel*
  - Type: *Label (JLabel from the Java Swing library)*
  - Description: *The Label for source code input.*
  - Constraints: *None*

- Attribute: *codeInputButton*
    - Type: *Button (JLabel from the Java Swing library)*
    - Description: *The button the user will interact with to open the file choose menu.*
    - Constraints: *None*
- Attribute: *fileChooser*
    - Type: *FileChooser (JFileChooser from the Java Swing library)*
    - Description: *Allows the user to choose a file for source code input*
    - Constraints: *codeInputButton has been clicked.*

4.2.6.2 Method Descriptions

CodeInputStrategy is an implementation of the ViewStrategy interface. The outlines of the methods for this class are outlined in the interface. See Section 4.3.6 for method descriptions.

4.2.7 Class: LMSInputStrategy

- Purpose: *View for selecting a target LMS for PPALMS problem.*
- Constraints: *See Section 4.3.6.*
- Persistent: *See Section 4.3.6.*

4.2.7.1 Attribute Descriptions

- Attribute: *lmsTargetLabel*
    - Type: *Label (JLabel from the Java Swing library)*
    - Description: *The Label for LMS target.*
    - Constraints: *None*
- Attribute: *lmsTargetComboBox*
    - Type: *ComboBox (JComboBox from the Java Swing library)*
    - Description: *User interactable combo box (drop-down menu) that provides LMS targets for selection.*
    - Constraints: *None*
- Attribute: *problemTypeLabel*
    - Type: *Label (JLabel from the Java Swing library)*
    - Description: *The Label for PPALMS problem type.*
    - Constraints: *None*
- Attribute: *problemTypeComboBox*
    - Type: *ComboBox (JComboBox from the Java Swing library)*
    - Description: *User interactable combo box (drop-down menu) that provides problem type selections based on the target LMS.*
    - Constraints: *A target LMS has been selected from the lmsTargetComboBox.*
- Attribute: *confirmButton*
    - Type: *Button (JButton from the Java Swing library)*
    - Description: *User interactable confirmation button. The user will press this when they have LMS targets for selection and indicated problem type.*
    - Constraints: *Will be disabled until an LMS target has been selected.*

4.2.7.2 Method Descriptions

LMSInputStrategy is an implementation of the ViewStrategy interface. The outlines of the methods for this class are outlined in the interface. See Section 4.3.6 for method descriptions.

4.2.8 Class: ProblemInputStrategy

- Purpose: *View for specifying details and exporting PPALMS problem.*
- Constraints: *See Section 4.3.6.*
- Persistent: *See Section 4.3.6.*

4.2.8.1 Attribute Descriptions

- Attribute: *titleInputLabel*
    - Type: *Label (JLabel from the Java Swing library)*
    - Description: *The Label for PPALMS problem title.*
    - Constraints: *None*
- Attribute: *titleInputTextField*
    - Type: *TextField (JTextField from the Java Swing library)*
    - Description: *User interactable text field for providing a title to the PPALMS problem.*
    - Constraints: *None*
- Attribute: *descriptionInputLabel*
    - Type: *Label (JLabel from the Java Swing library)*
    - Description: *The Label for PPALMS problem description.*
    - Constraints: *None*
- Attribute: *descriptionInputTextField*
    - Type: *TextField (JTextField from the Java Swing library)*
    - Description: *User interactable text field for providing a description to the PPALMS problem.*
    - Constraints: *None*
- Attribute: *annotationsInputLabel*
    - Type: *Label (JLabel from the Java Swing library)*
    - Description: *The Label for PPALMS annotations field.*
    - Constraints: *None*
- Attribute: *sourceCodeArea*
    - Type: *TextArea (TBD component from Java Swing library)*
    - Description: *Text area that will be user-interactable for adding annotations to the given PPALMS problem.*
    - Constraints: *None*
- Attribute: *exportButton*
    - Type: *Button (JButton from the Java Swing library)*
    - Description: *User interactable confirmation button. The user will press this when they have completed their problem.*
    - Constraints: *Will be disabled until at least one problem annotation has been added.*

4.2.8.2 Method Descriptions

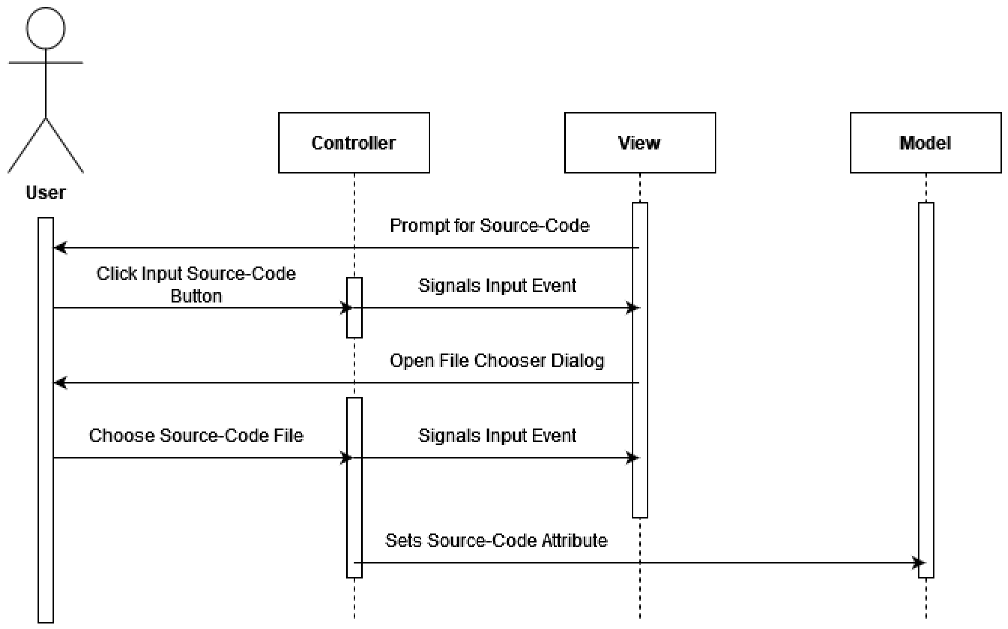
ProblemInputStrategy is an implementation of the ViewStrategy interface. The outlines of the methods for this class are outlined in the interface. See Section 4.3.6 for method descriptions.

5 Dynamic Model

The purpose of this section is to model how the system will respond to events by the user. The following section provides scenarios that will display how the user will interact with the system and how the components of the system will interact together.

5.1 Scenario #1 - Uploading Source Code

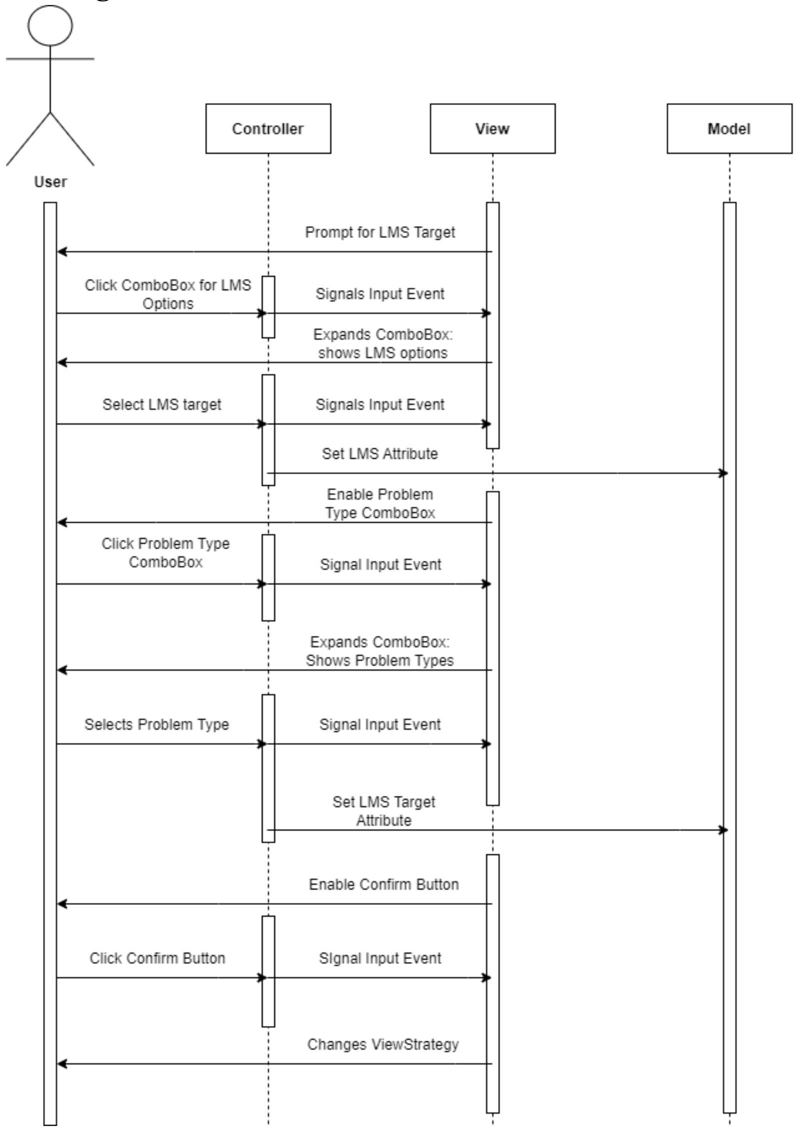
- **Scenario Description :** Upon starting the application, the user will be prompted to upload their source code to begin creating their problem. The user will click the input source code button, which will trigger an event to open the file chooser. The user will then select their source code file from their file system.
- **Scenario Diagram :**



Sequence Diagram for Uploading Source Code

5.2 Scenario #2 - Selecting Target LMS and Indicate Problem Type

- **Scenario Description :** The view displays a prompt for the LMS Target selection to the user. The user is then allowed to click on a combobox for LMS options. This signals an event from the controller to the View, displaying the expansion of the ComboBox along with the LMS options. The user then clicks on the option from the ComboBox, which the controller then signals an input event to view. Finally, the set LMS attribute is relayed to the Model. After the target LMS has been selected, the input for problem type will become available. Following the similar steps as the LMS target, the user will select a problem type from the problem type ComboBox. The user will proceed by selecting the confirm button, which will be available for use when the user has indicated both the target LMS and the problem type.
- **Scenario Diagram :**

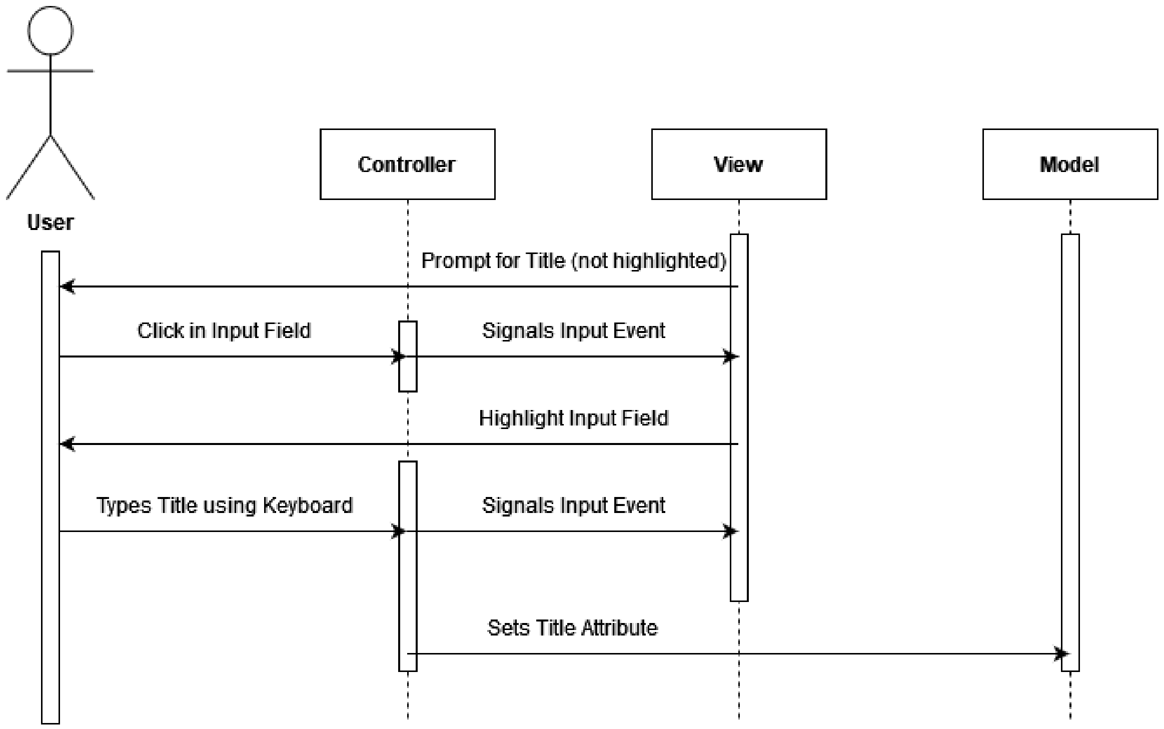


Sequence Diagram for Selecting Target LMS

5.3 Scenario #3 - Entering Title (Optional)

NOTE : *The scenario is "Optional", as it's not required for the user to enter a title for Parson's Problems. This also applies to the scenario where a user were to provide a Description.*

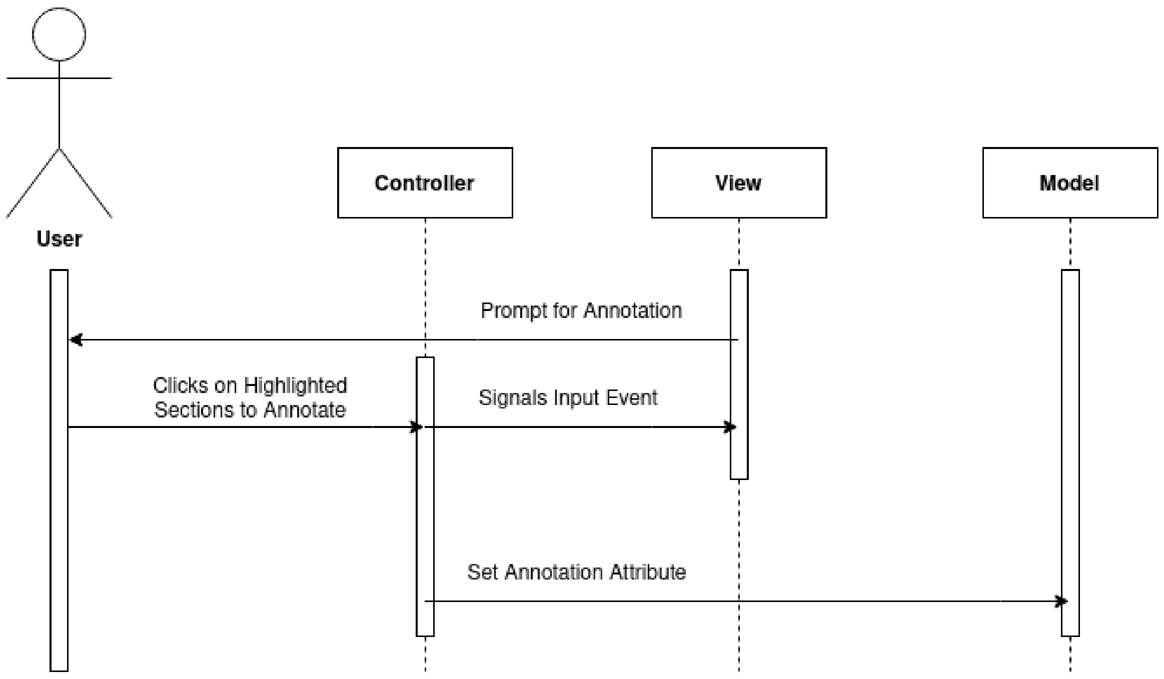
- **Scenario Description :** When a user clicks on the textbox input field for the title, it signals an input event from the controller to the view to highlight the field, indicating to the user to provide a title. The user then proceeds by providing a relevant title for the Parson's Problem. When they click away from the textbox field, it further signals an input event from controller to view on its completion and the controller sets the title attribute and sends it to the model.
- **Scenario Diagram :**



Sequence Diagram for Entering Title

5.4 Scenario #4 - Create Problem Annotation

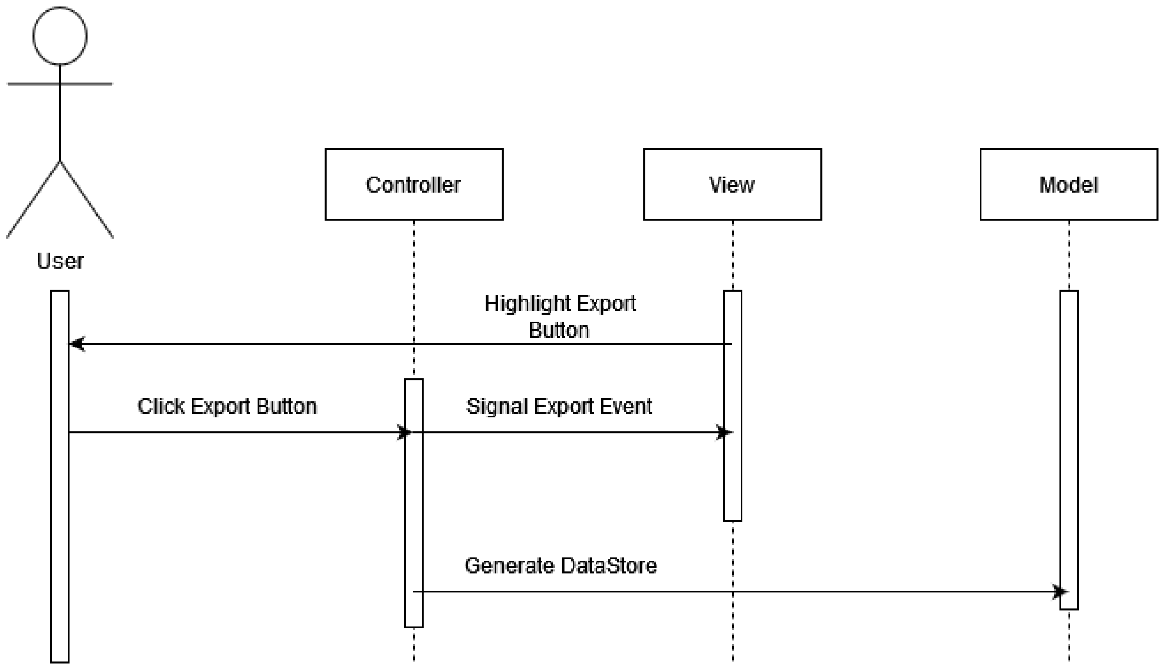
- **Scenario Description :** Upon the system's completion in generating the Parson's Problems, it will then prompt the user to perform annotations on the problems. The system indicates this by highlighting elements of the problem when the user's mouse cursor hovers above them. The user may then select the elements by clicking on them, signaling input events to the controller which is reflected in the view.
- **Scenario Diagram :**



Sequence Diagram for Annotating Parson's Problem

5.5 Scenario #5 - Export Parson's Problem to LMS

- **Scenario Description :** The scenario would take place when all required fields have been filled in, which is when the export button is highlighted. The user is then able to click on the export button, triggering a signal export event to the controller and reflected in view as a progress bar. In the background, PPALMS generates a DataStore (*refer to section 3 of the document for more information on what a DataStore is and its purpose*), which is type specific based on the selected target LMS that it will be exported to.
- **Scenario Diagram :**

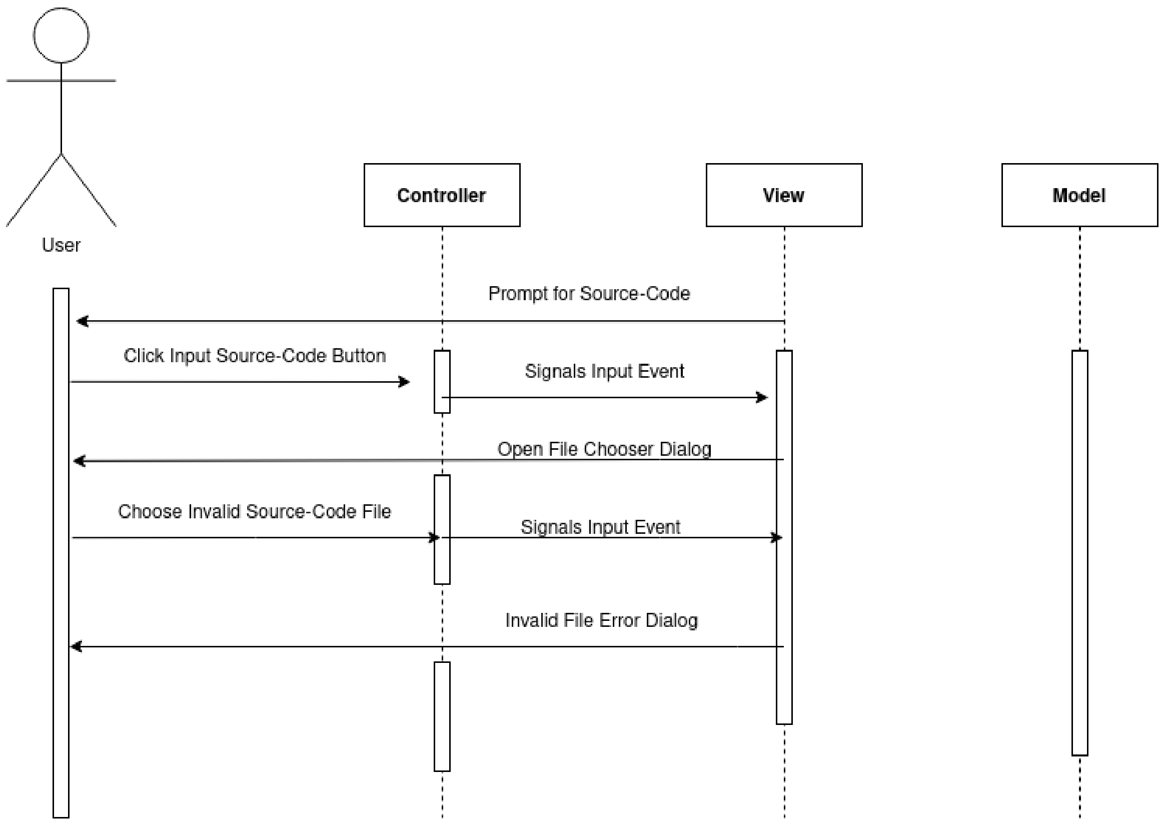


Sequence Diagram for Exporting Parson's Problem

5.6 Scenario #6 - Incorrect Source-Code File Input

NOTE : *This scenario is different to prior scenarios as it's an error scenario, and depicts how the PPALMS would react to invalid input. Similar error scenarios relating to input invalidity will be responded to in a similar manner.*

- **Scenario Description :** PPALMS prompts the user to provide a source-code file which is enabled by clicking the input source-code button, signaling an input event from controller to view. An open file chooser dialog is displayed by view, indicating to the user of the action. Now, suppose if the user provides an invalid source-code file, the PPALMS would signal an input event from controller to view, but instead of processing the contents of the file, an invalid file error dialog is displayed to the user instead.
- **Scenario Diagram :**



Sequence Diagram for Incorrect Source-Code File Input

6 Nonfunctional Requirements

One nonfunctional requirement of the PPALMS outlined in the SRS is to have sufficiently fast performance. This is achieved through the widely-used Java Swing dependency for implementation of the GUI . Using Java for the application allows for sufficiently fast processing times, and using Java Swing for the GUI allows for sufficiently fast response times with input interactions.

Another nonfunctional requirement of the PPALMS outlined in the SRS is to be safe and secure. Since the implemented design allows the user to interact only through preconfigured and sanitized interactions, and stores problem information dynamically, it reduces the likelihood of malicious activity or stolen data to an acceptable level. Even in the case of uncontrollable behavior like program crashes or power loss, local information about a problem would be lost and required to be entered once more. The responsibility to secure problems after creation is offset to the target LMS.

The next nonfunctional requirement of the PPALMS outlined in the SRS is to be adaptable, reusable and open to extension to accommodate various file types, LMS targets, and Parson's problem types. The design outlined in this document achieves this by using a solid design principle of the Strategy pattern. This allows for different strategies to be injected into the architecture to accommodate different file types, LMS targets, and Parson's problem types which could not be specified or foreseen by this document.

Another nonfunctional requirement of the PPALMS outlined in the SRS is to make Parson's Problem creation accessible to a general user. The MVC architecture outlined in this document achieves this by abstracting the problem creation process into Model , View, and Controller modules, and hiding their internal components from the User. The specifics of how the problem is stored and edited, as well as how input is expressed and handled is hidden from the user, and not required of them to know. The User's interactions are constrained to a more human-readable and understandable interface which a general user can be expected to navigate with the View GUI in Java Swing, and hence the requirement is satisfied by the outlined design.

Some nonfunctional requirements outlined in the SRS are not under the control of the design outlined in this document. Parameters like the performance of the machine running the PPALMS application or external target LMS servers, performance of peripheral devices which display the GUI or take in input/output, competency of general users, and source code quality or content are not within the scope of the PPALMS system to control. Using the most optimal of what is feasible and effective error response is the best way to satisfy these requirements.

7 Requirements Traceability Matrix

This section provides a mapping of how the design elements relate to the PPALMS SRS document. Here, we have analyzed design elements such as the interfaces, data stores, classes (or attributes and methods if the class is too general), as well as scenarios and how these elements relate to the external, system, and nonfunctional requirements. We have formatted this mapping as a traceability matrix to better illustrate the correlation between the design document and the SRS document. SRS requirements are the column headers and the design document elements are the row headers. A zero/blank value indicates that no relationship between the two documents exists, while an x indicates mapping between the two items.

SRS 2.1, which is the User Interface External Requirement, is covered by the Application Interface section in 3.1.1, the PpalmsGUI class in 4.2.4, and ViewStrategy class in 4.2.5 from the design document

SRS 2.2, which is the Hardware Requirement, is covered by the Data Stores section in 3.2 of the design document because both talk about using files, which is a type of data store.

SRS 5.1, which is the Inputting Source Code Requirement, is covered by the PpalmsProblem class in 4.2.1, the PpalmsLogicHandler class in 4.2.2, the PpalmsInputHandler class in 4.2.3, and the CodeInputStrategy class in 4.2.6 of the design document. It also is covered by Scenario #1 - Uploading Source Code in section 5.1 as well as Scenario #6 - Incorrect Source-Code File Input in section 5.6 of the design document.

SRS 5.2, which is the Selecting Target LMS Requirement, is covered by the PpalmsProblem class in 4.2.1, the PpalmsInputHandler class in 4.2.3, and the LMSInputStrategy class in 4.2.7 of the design document. It is also covered by Scenario #2 - Selecting Target LMS in section 5.2 of the design document.

SRS 5.3, which is the Selecting Parson's Problem Type Requirement, is covered by the PpalmsInputHandler class in 4.2.3, and the ProblemInputStrategy class in 4.2.8 of the design document.

SRS 5.4, which is the Including/Excluding Annotations Requirement, is covered by the PpalmsProblem class in 4.2.1, the PpalmsLogicHandler class in 4.2.2, and the PpalmsInputHandler class in 4.2.3. It is also covered by Scenario #4 - Create Problem Annotation in section 5.4 of the design document.

SRS 5.5, which is the Providing Optional Title and Description Requirement, is covered by the PpalmsProblem class in 4.2.1, the PpalmsLogicHandler class in 4.2.2, and the PpalmsInputHandler class in 4.2.3. It is also covered by Scenario #3 - Entering Title (Optional) in section 5.3 of the design document.

SRS 5.6, which is the Validation for File Output Requirement, is covered by the PpalmsLogicHandler class in 4.2.2 as well as Scenario #5 - Export Parson's Problem to LMS in section 5.5 of the design document.

SRS 5.7, which is the Generating Permutations Requirement, is covered by the PpalmsLogicHandler class in 4.2.2 as well as Scenario #5 - Export Parson's Problem to LMS in section 5.5 of the design document.

SRS 6, which are the Nonfunctional Requirements, is covered by the Nonfunctional Requirements in section 6 of this design document.

	(SRS 2.1)	(SRS 2.2)	(SRS 5.1)	(SRS 5.2)	(SRS 5.3)	(SRS 5.4)	(SRS 5.5)	(SRS 5.6)	(SRS 5.7)	(SRS 6)
3.1.1	x									
3.2		x								
4.2.1			x	x		x	x			
4.2.2			x			x	x	x	x	
4.2.3			x	x	x	x	x			
4.2.4	x									
4.2.5	x									
4.2.6			x							
4.2.7				x						
4.2.8					x					
5.1			x							
5.2				x						
5.3							x			
5.4						x				
5.5								x	x	
5.6			x							
6										x